Productivity in Parallel Programming: A Decade of Progress

Looking at the design and benefits of X10

John T. Richards, IBM T. J. Watson Research Center

Jonathan Brezin, IBM T. J. Watson Research Center

Calvin B. Swart, IBM T. J. Watson Research Center

Christine A. Halverson, IBM Almaden Research Center

In 2002 DARPA (Defense Advanced Research Projects Agency) launched a major initiative in HPCS (high-productivity computing systems). The program was motivated by the belief that the utilization of the coming generation of parallel machines was gated by the difficulty of writing, debugging, tuning, and maintaining software at peta scale.

As part of this initiative, DARPA encouraged work on new programming languages, runtimes, and tools. It believed that by making the expression of parallel constructs easier, matching the runtime models to the heterogeneous processor architectures under development, and providing powerful integrated development tools, it might improve programmer productivity. This is a reasonable conjecture, but we sought to go beyond conjecture to actual measurements of productivity gains.

While there is no established method for measuring programmer productivity, it is clear that a productivity metric must take the form of a ratio: programming results achieved over the cost of attaining them. In this case, *results* are defined as successfully creating a set of parallel programs that ran correctly on two workstation cores. This is a long way from peta scale, but since new parallel software often starts out this way (and is then scaled and tuned on ever-larger numbers of processors), we viewed it as a reasonable approximation. Moreover, results found with two cores should be of interest to those coding nearly any parallel application, no matter how small. *Cost* was simpler to handle once results was defined, since it could reasonably be approximated by the time it took to create this set of parallel programs.

The purpose of this study was to measure programmer productivity, thus defined, over several years starting in 2002, the beginning of the HPCS initiative. The comparison was primarily focused on two approaches to parallel programming: the SPMD (single program multiple data) model as exemplified by C/MPI (message-passing interface), and the APGAS (asynchronous partitioned global address space) model supported by new languages such as X10 (<http://x10-lang.org>), although differences in environment and tooling were also studied. Note that the comparison was *not* between C/MPI as it has come to be and X10 as it is now. Rather, it was a historical contrast of the way things were in 2002 with the way things are now. Indeed, C++ with its exceptions and MPI-2 with its one-sided communication protocol likely enhance programmer productivity and are worthy of study in their own right.

Given our objective, we sought to replicate as closely as possible the programming environment found in 2002 for users of C/MPI. This included the gdb debugger, along with a typical set of command-line tools. For X10, we used Eclipse (<http://www.eclipse.org>) with the X10 plug-in as it was found in 2010, the date of this study. X10 was developed as part of the HPCS initiative. It combines

the succinctness of newer languages such as Scala (<http://www.scala-lang.org>) with a model of concurrent programming that maps nicely to the PGAS model of modern parallel machines. After a decade, the language is still evolving, but the basic model was stable when this study began. Importantly, no X10 debugger was available until after this study was completed, so current users of X10 can expect to see gains even larger than those reported here.

PRIOR WORK

This is not the first study to attempt to measure the productivity of programming in X10. Ebcioglu *et al.*⁵ and Danis and Halverson³ describe an experiment in which undergraduates (novices with respect to parallel programming) were trained in either C/MPI, UPC (Unified Parallel C), or (a very early version of) X10 and then attempted to parallelize the string-matching algorithm of SSCA 1 (Scalable Synthetic Compact Application 1), as described in Bader *et al.*² Average time to completion was approximately 510 minutes in C/MPI, 550 minutes in UPC, and 290 minutes in X10. When the differences in code-execution time during testing were removed, these times were approximately 360 minutes for C/MPI, 390 minutes for UPC, and 180 minutes for X10. Interpretation of this roughly twofold productivity gain was somewhat complicated, however, by the absence of clear success criteria for code completion.

In a subsequent study, Halverson and Danis⁶ added the Eclipse programming environment to the tools available to X10 programmers. Participants in this study were more experienced, having had some parallel programming training in earlier coursework. A productivity gain for X10 and Eclipse over C/MPI was found here as well, but computing the size of this gain was complicated by the fact that only one of the seven C/MPI participants completed the task (in 407 minutes) vs. five of the seven X10 participants in an average of 321 minutes.

In both of these earlier studies, only one program was coded (a portion of SSCA 1), only relatively novice parallel programmers were tested, and the task was limited to parallelizing an embarrassingly parallel algorithm that was provided to the test participants in working serial code. All three of these deficiencies are addressed in the assessment reported here.

THE EXPERIMENTAL APPROACH

In a long-running study we measured the time to develop six parallel programs in each of two languages, for a total of 12 programs in all. Each program was developed from a problem statement and/or algorithm description without benefit of any preexisting code samples. The time to completion ended with the first successful parallel run using both cores of a two-core workstation.

Two of the six programs were defined in phase 2 of the DARPA HPCS program, SSCA 1 and SSCA 2, and are described in Bader *et al.*² SSCA 1 is the string-matching problem used in the two previously referenced studies. In this study, however, two versions of SSCA 1 were developed in each language: the embarrassingly parallel version that works well when the target string is much shorter than the search string, and a more complex anti-diagonal version that is better suited for cases where the two strings are of roughly equal length. SSCA 2 was new to this assessment, with all four kernels being coded in each language.

In addition to these two SSCAs, four more problems were defined, based on an analysis of the Berkeley motifs.¹ All six problems are described in the next section.

Developing these 12 programs took nearly a year. As desirable as it would be to have multiple

skilled programmers perform this task, it would have greatly exceeded available funds. Instead, study costs were contained by having only a single programmer—a member of our productivity assessment team and one of the authors of this report—serve as the test participant. A Ph.D. mathematician by training, he has been programming in C professionally since 1979, wrote the front end of IBM's first C compiler, and has been writing MPI for multicore systems since 2007. He began programming X10 in 2007 as well, developed the Halverson and Danis⁶ study test bed, and wrote several sections of the X10 tutorial. While it is tempting to view the results from a single programmer as anecdotal, the observed productivity gains were within the range of those found for novice programmers on simpler tasks and are likely quite conservative given the programmer's greater proficiency with C and the lack of an X10 debugger during the study period.

THE PROGRAMS

Six programming problems were defined for this study, each representing a class of parallel application. Collectively, they span a substantial range of typical small parallel programs:

- **SSCA 1 (first kernel).** This involves a string-matching problem that was motivated by chromosome and protein studies. One is asked to find the best approximate matches between substrings of a pair of strings of uppercase letters.
- **SSCA 2 (all four kernels).** This problem involves the efficient representation of a sparse graph whose edge set is known *a priori* to be randomly distributed across the set of available processors. The first kernel constructs the graph in a format to be used as input by all subsequent kernels. The second kernel extracts edges by weight from the graph representation and forms a list of the edges with maximal weight. The third extracts a series of subgraphs formed by following paths of specified length from a starting set of initial vertices. The final kernel computes a network centrality metric that identifies vertices of key importance along the shortest paths of the graph.
- **Consumer-Producer.** This problem uses one process as a server managing a queue for multiple client processes that randomly add to and remove from the queue. Since one buffer is the target for all of the operations, the first programming problem is to minimize contention. Since adding elements to and removing elements from the queue occur at opposite ends, this is the main opportunity for avoiding contention; you can safely do these two operations simultaneously if the queue is large enough.
- **Unbalanced Tree Search.** In contrast to the previous problem in which coordination is managed through a central server process, here a set of peer processes each manages its own queue of work and, as the need arises, shares work with other processes. The overall task is a breadth-first search of a tree that is known to have leaves at highly varying depths from the root. Each processor is managing its own queue of unvisited tree nodes but may be asked by an idle peer to provide some nodes for it to pursue.
- **Floyd's Algorithm.** Given a directed graph with weighted edges that is either acyclic or has only non-negative weights, we compute the summed weights of the shortest paths between all pairs of vertices.
- **Discrete Fourier Transform.** The algorithm implemented is the original Cooley-Tukey Fast Fourier Transform. The input is an array of 2^n complex numbers, as is the output. Until at least $n=3$ and $2^n=8$, the conventional serial algorithm beats Cooley-Tukey, which means that you can profitably distribute the array across $2^n/8$ processes.

OBSERVED PRODUCTIVITY GAIN

Table 1 summarizes the number of days to first successful parallel run and the number of lines of code written for each of the 12 programs. The six programs in X10 required a total of 39 days to develop. The six programs in C with MPI required a total of 129 days to develop. Overall productivity gain due to language (and, secondarily, environment) was, therefore, in excess of 3x. Over the 39 days of writing X10, 6,195 lines of code were written at an overall rate of 159 lines of code per day. Over the 129 days of writing C with MPI, 10,245 lines of code were written at an overall rate of 79 lines of code per day. While we did not measure the performance of these programs, a study by Saraswat *et al.*⁷ examined the excellent parallel performance of an X10 implementation of an unbalanced tree search program quite similar to ours.

PRODUCTIVITY CONTRIBUTORS

What accounts for the threefold productivity gain found in this study? While some of the gain is a result of the integrated tooling provided by Eclipse and the X10 plug-in, most is attributable to features of the X10 language. These include the task and data partitioning provided by `activity` and `place`, the flexible task support provided by `async` and `finish`, and a nicely integrated exception-handling mechanism. In combination these simplified the expression of parallelism and allowed the program structure to mirror the natural problem structure more closely. In addition, automatic garbage collection removed the chore of managing memory, and object orientation simplified the task of dealing with complex structures while avoiding the miscalculation of memory references. The following sections provide examples of how these features contributed to productivity.

EXPRESSING PARALLELISM

X10 programs begin with a single activity executing at a single place. An activity is a single sequential thread of execution, and a place is a piece of the system that combines processing power with a share of the partitioned global memory. Any activity can, as needed, initiate new activities at the place where it is currently running or at any other convenient place.

Local threading. As a simple example, SSCA 1 requires reading a pair of character strings from some peripheral source. If the strings are very long (as they might well be since this problem is designed to scale to arbitrarily long strings), then it is desirable to read the two concurrently. If both streams are to be accessed from a single processor, the X10 code you need is shown in figure 1.

Simple fork-join parallelism is done in X10 using an `async` statement to do the fork, and

TABLE 1 Days to first successful parallel run on two cores and lines of code in completed programs

Programs	X10 Days	X10 Lines	C/MPI Days	C/MPI Lines
SSCA 1	9.0	1625	20	1225
SSCA 2	7.9	1150	28	3000
Consumer-Producer	5.0	1275	24	1750
Unbalanced Tree Search	7.5	1125	30	2200
Floyd's Algorithm	3.8	520	12	1400
Discrete Fourier Transform	5.6	500	15	670
Total	38.8	6195	129	10,245

surrounding the set of activities with a `finish` block to implement the join. The `try` block that surrounds the whole computation catches any errors that might have occurred in the file I/O or activity startup/shutdown.

For C, the essential problem is how to get a second thread with which the originating thread can share memory. Figure 2 shows the same code as it might be done in C using POSIX threads.

The SPMD model's relatively bad fit surfaces in the test for `my_id==0`, a typical C/MPI idiom. If an I/O error occurs, processor 0 has to let the other processors know there was a problem. That is why there *must* be a broadcast after the strings have been read in, whether or not there were any problems. The function `do_the_read` encapsulates the second thread's mission. Even though it is doing exactly the same read as the root thread, it must be separate.

Remote threading. The previous example is so simple that it is easy to underestimate its significance. To get a better sense, consider a slight variation. Suppose that, instead of processor 0 being known to handle most of the work, the processors where the data resides are known only at runtime, as they might be if the data is coming in from a network or from a set of databases. How does the code change? Figure 3 shows the X10 code.

Places in an X10 program, like MPI processes, have a unique integer ID. In figure 3 the two sequences are being read from the places with IDs `seq1Id` and `seq2Id`. The read operations differ from the original only in that an `at` clause has to be provided to say where each read is to take place. All the remaining code is unchanged, including the error handling, *even though multiple processors might now be involved*. This is a consequence of X10's exception model, which is designed to mirror a program's activity tree. It allows any activity in that tree to handle exceptions thrown by anything in the subtree below it. (An X10 exception does not automatically include the place it was thrown, although a programmer can readily subclass an exception and insert place information. Also, exceptions have no impact on the children or siblings of the activity in which the exception occurs.) Although C++ introduced the `try/catch` style of exception handling for serial code before 2002, none

FIGURE 1

X10 Code Needed to Read Two Streams in Parallel

```
try {
    val seq1: Array[Byte](1); // seq1 and seq2 are arrays of bytes
    val seq2: Array[Byte](1); // indexed by a single integer
    finish { // control leaves only when both activities complete
        async{ seq1 = readASequence(path1); } // spawns an activity
        seq2 = readASequence(path2); // runs in parallel
    }
    compare(seq1, seq2); // both sequences are available here
}
catch(e: Exception) { handleTheProblem(e); }
```

of the threading frameworks carries it over to sets of multiple processes with the same simplicity of X10 in which both single- and multi-threaded cases are supported without change.

Access Serialization: A single-server queue is a common pattern where serial writes to a stream of items by one set of players are read by a possibly different set. The usual solution is to buffer the items in an array until they are consumed. Items are added at the end of the array but removed from its beginning.

With multiple players, the potential for race conditions in this scenario is obvious. Remove

FIGURE 2

C Code Needed to Read Two Streams in Parallel

```
typedef struct {
    size_t size; /* of the data array in bytes */
    int rc;      /* return code for the i/o, normally 0 */
    char buffer[]; /* either the string or an error message */
} ByteArray;

static void do_the_read(void *arg) {
    char *path = (char *)arg;
    pthread_exit(read_sequence(path));
}

...

int rc;
if (my_id == 0) { /* am I the processor with MPI rank 0? */
    pthread_t thread; /* yes, so I am in charge */
    ByteArray *seq1, *seq2;
    rc = pthread_create(&thread, NULL, do_the_read, path1);
    if (rc != 0) perror("Thread creation failed");
    else {
        seq2 = read_sequence(path2);
        rc = pthread_join(thread, (void *)&seq1);
        if (rc || seq1->rc || seq2->rc) {
            rc = report_the_problem(rc, seq1, seq2);
        }
    }
}
MPI_Bcast(&rc, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (rc == 0) return compare(seq1, seq2);
else return rc;
```

operations from the array must be serialized with respect to one another, as must adds. On the other hand, unless few items remain, an add and a remove need not be serialized with respect to one another, because they affect opposite ends of the array.

How might one wring out the last bit of performance by allowing an add in parallel with a remove when that is safe? Serializing two adds or two removes needs only a lightweight atomic version of C's postfix `++`. When few items remain, all of the operations have to be serialized, which requires an atomic block of code to guarantee that a thread, once active in the block, will be the only thread in the block until it exits the block.

In X10 the class `AtomicInteger` provides atomic update to a single integer-valued variable. Because this uses hardware support, it is a much lighter-weight operation than protecting a whole block. The keyword *atomic* in front of a block of statements serializes access to the block. X10 thereby provides what is needed to write 650 lines of code that read quite naturally. As of 2002, there was no standardized C API analogous to either `AtomicInteger` or atomic blocks. (See <http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html>. This API is essentially what was standardized in C11.) The serialization you need from both atomic blocks and atomic updates can be achieved by using synchronous MPI calls such as `MPI_Recv`, but the resulting code is longer—1,100 lines in our implementation—and considerably harder to follow.

For those wondering what could take 1,100 lines, only a very small part of the code supports the queue operations. This is a client-server application. The server has to start, it has to stop, clients have to make requests, there has to be access control (not everyone can use every array), and so on. What makes the X10 version so much shorter is simpler error handling and not having to write both a client and a server API.

Peer Coordination. In the previous example, coordination is managed through a central server process. What happens when a set of peer processes each manages its own queue and, as the need

FIGURE 3

X10 Code to Read using Processors Known Only at Runtime

```
try {
    val seq1: Array[Byte](1); // seq1 and seq2 are arrays of bytes
    val seq2: Array[Byte](1); //      indexed by a single integer
    val seq1Home = Place.place(seq1Id); // the id names a Place object
    val seq2Home = Place.place(seq2Id); // that reifies the processor
    finish {
        async { seq1 = at(seq1Home) readASequence(path1); }
        seq2 = at(seq2Home) readASequence(path2);
    }
    compare(seq1, seq2);
}
catch(e: Exception) { handleTheProblem(e); }
```


arises, shares items with other processes? Here each process acts as both client and server for some subset of the active processes. The example we programmed performs a breadth-first search of a tree that is known in advance to be substantially unbalanced. Each processor manages its own queue of unvisited nodes, but, to spread the workload more evenly, a processor may be asked to ship some unhandled nodes to another processor to work on.

The biggest issue here is termination: how does a process know when not only it is out of work, but all the other processes are also out of work? Algorithms to handle this (at least at lower orders of scale) date back some 30 years to Dijkstra *et al.*⁴ and have been refined a number of times since. The SPMD solution is based on them. The X10 solution is entirely different, relying on threading on demand.

Let's look at X10 first. A single X10 activity begins the action by asking each peer in a set to start searching a part of the tree. If one of these peers runs out of work, it has a list of "nearby" peers it is permitted to ask for more work. It does this by sending a reference to itself to each of its neighbors and then lets its current activity die. The *object* sending these requests, however, continues to exist. Thus, if one of the idle peer's neighbors has work to spare, it can spawn a new activity at the idle peer's place in which that peer can resume work. A little handshaking is involved if the peer wants to make sure that only one neighbor's work is taken on, but otherwise that is the whole story. Since the root activity has spawned the multiple peer activities as *asyns* within a single *finish* block, it can merely exit that block when none of these *asyns* remains active, thereby finishing the job.

The C solution is entirely different. There is no peer object at each processor ready to receive a remote process call, and even if there were, there is no analog of the X10 *finish* block. The naïve way of handling an idle peer is to let it hang on a synchronous receive, but then who is responsible for knowing when to send that peer the "all done" message? That is exactly the problem Dijkstra *et al.* were addressing by a clever method of polling all the peers.

In the C solution each processor must perform three activities: the primary work (namely, searching and processing its share of the tree); listening for a termination control message; and listening to its neighbors for shared work requests. By using two nonblocking receives to listen for the control and sharing communications, each processor has, in effect, three concurrent activities: the primary activity and the two pending receives. Each processor's main loop is responsible for dealing with incoming control and sharing requests in a timely manner. Most of the complexity is involved in deciding who sends what control information to whom.

MANAGING MEMORY AND HANDLING ERRORS

The issues discussed in the previous section pertain to the fit between the natural threading models of our programs on the one hand and the X10 APGAS and MPI SPMD parallel models on the other. There are also several well-known C-related shortcomings that impacted our productivity in writing serial sections of code. These shortcomings are not present in X10. On average, we encountered about six of these well-known problems per 1,000 lines of C code, and the overall impact was significant. Many required additional effort because they did not reveal themselves close to the point where a correction was needed.

Memory leaks. X10, like many recent languages, has automatic garbage collection. The need to explicitly manage storage has long been recognized as one of the serious impediments to C coding productivity. You might think that in a problem as simple as, for example, SSCA 1, with around

1,200 lines of relatively straightforward string-matching code, this would not be an issue—but it is, especially if the code is to run for a long time (or be incorporated into a frequently called library routine). Here memory leaks need to be found and eliminated, a task that can easily take one or two days for the places found in our code where memory was allocated but never freed.

Getting memory references right. The lack of introspection in C is a significant problem as well. In an object-oriented language such as X10, objects know how big they are and what outstanding references exist. In C those calculations must be done by hand. It is not that, using X10, you cannot miscompute an array extent. It is still possible, but X10 will catch an out-of-bounds access to an array at runtime at the point of the access. Even skilled C programmers are unlikely to take the time to bulletproof every array access. Since errant accesses are usually detected far from the point where the code needs to be corrected, these errors are not easily found and fixed.

Error handling. The lack of a convenient exception mechanism in C forces programmers to be more verbose. This surfaced in Floyd's algorithm, for example, when our programmer wanted a generic input stream to read an ASCII stream of numeric values. His API had an entry that tokenizes the stream, converts the tokens to the appropriate numeric type, and assures that the value is legal. Clearly there are a number of problems that the stream can encounter. The question is how to handle these errors.

In the case of errors in X10, an exception is thrown whose type identifies the problem encountered. An application can avoid providing special code for detecting generic errors such as an unexpected end-of-file that is discovered by lower-level functions, because they, too, can signal errors by throwing exceptions. The application can therefore concentrate on signaling semantic errors in the content.

For most throwaway code, error handling is not a serious issue, but for production code, and for complex communications patterns such as Dijkstra *et al.*'s termination algorithm, it certainly is. C's signaling mechanism is best suited for expert use. C's problems, however, run even deeper in a multithreaded SPMD world. Consider the standard library routine `strtoll` that the stream calls to convert a found token to a long integer. Here is the discussion of `strtoll`'s error indications as given by its "man" page:

"the `strtol`, `strtoll`... functions return the result of the conversion, unless the value would underflow or overflow. If no conversion could be performed, 0 is returned and the global variable `errno` is set to `EINVAL` (the last feature is not portable across all platforms). If an overflow or underflow occurs, `errno` is set to `ERANGE`..."

Consider the code a C application needs in order to deal with the various possible errors. Should the code make sure to zero `errno` before calling `strtoll`? After all, it might be non-zero because of an entirely unrelated earlier problem. For the code that checks `errno` to understand what happened, it is, moreover, not enough just to check that `errno` is non-zero, because the error may be an I/O error, a parsing error, or a range error. Nor can you be sure that `errno` is thread-safe—it is not on all systems. What then? And where in the application should you clear `errno`, whose value is a global? Which of the other processes need to be made aware of the problem, and how should they be made aware?

CONCLUSION

There are very good reasons for C and MPI's dominance in the parallel-programming community. They are superbly documented, elegant, clean designs that have been carefully implemented. In the hands of an experienced, disciplined professional, they provide a level of control that is simply not available elsewhere. Neither C nor MPI is a standing target: both continue to be improved, even though they are now mature technologies.

How often, however, do the benefits of C/MPI outweigh its costs? Through all three of our studies, and particularly in this final one, we have seen substantial benefits—ranging from two to six times faster development to first successful parallel run—from using a higher-level language and programming model. These productivity benefits might be even greater for large programs that need to be maintained for years.

X10 and its APGAS programming model are now being explored in many research and university settings. While the language may or may not cross over into mainstream use, it is likely that the qualities that made it so much more productive for us will likely become well established in the parallel community: flexible threading, automatic garbage collection, runtime type-driven error checking, partitioned global memory, and rooted exception handling are all valuable. We hope that our experiments encourage those looking to improve parallel-programmer productivity to seriously study X10's design and its benefits.

ACKNOWLEDGMENTS

This work was supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. The authors would like to thank Catalina Danis, Peter Malkin, and John Thomas, members of IBM's HPCS productivity assessment team, for their invaluable contributions to this research.

REFERENCES

1. Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. QW, Yelick, K. A. 2006. The landscape of parallel computing research: a view from Berkeley. Technical Report No. UCB/EECS-2006-183. Electrical Engineering and Computer Sciences, University of California at Berkeley; <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf>.
2. Bader, D. A., Madduri, K., Gilbert, J. R., Shah, V., Kepner, J., Meuse, T., Krishnamurthy, A. 2006. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *CTWatch Quarterly* 2(4B); <http://www.cse.psu.edu/~madduri/papers/SSCA-CTWatch06.pdf>.
3. Danis, C., Halverson, C. 2006. The value derived from the observational component in an integrated methodology for the study of HPC programmer productivity. In *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing*: 11-21.
4. Dijkstra, E. W., Feijen, W. H. J., van Gasteren, A. J. M. 1983. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters* 16 (5): 217-219.
5. Ebcioğlu, K., Sarkar, V., El-Ghazawi, T., Urbanic, J. 2006. An experiment in measuring the productivity of three parallel programming languages. In *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing*: 30-36.
6. Halverson, C. A., Swart, C., Brezin, J., Richards, J., Danis, C. 2008. Towards an ecologically valid

study of programmer behavior for scientific computing. In *Proceedings of the First Workshop on Software Engineering for Computational Science and Engineering*.

7. Saraswat, V. A., Kambadur, P., Kodali, S., Grove, D., Krishnamoorthy, S. 2011. Lifeline-based global load balancing. *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*: 201-212.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

JOHN RICHARDS is a research manager in IBM's Watson Group and holds an appointment as Honorary Professor in the School of Computing at the University of Dundee, Scotland. He received the Alexander C. Williams Jr. Award from the Human Factors Society, and is a Fellow of the Association of Computing Machinery, a member of the IBM Academy of Technology, and a Fellow of the British Computer Society.

JONATHAN BREZIN was trained as a mathematician and held positions at the University of Minnesota and the University of North Carolina in the 1960s and 1970s, winding up as professor of mathematics at UNC in Chapel Hill in the mid-'70s. After short spells at Bell Laboratories and on Wall Street, he joined IBM Research. Initially he worked on optimizing compilers, but more recently he turned to supporting various accessibility initiatives. This work marks a return to his first love and his swan song—he retired from IBM this year.

CAL SWART joined IBM Research in 1982 and is currently a senior technical staff member in the IBM Watson Group. He graduated from Calvin College and has a degree in math and physics. He has worked on numerous projects during his career at IBM involving programmer toolkits, enhanced Web experiences, accessibility, productivity, and handheld applications. He is a member of Watson Life Research, exploring new applications of cognitive computing.

CHRISTINE HALVERSON is an independent consultant in Silicon Valley. Formerly she worked at IBM Research, where she spent five years working on the DARPA HPCS initiative studying parallel programmers. Other interests include CSCW (computer-supported cooperative work), HCI (human computer interaction), and applications of speech technology in a wide range of settings.

© 2014 ACM 1542-7730/14/0900 \$10.00